

UNITED STATES PATENT APPLICATION  
FOR

COMMAND LINE INTERFACE  
ABSTRACTION ENGINE

INVENTORS:

Mike J. Little, Stan Froyd

Prepared by:

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP

12400 WILSHIRE BOULEVARD

SEVENTH FLOOR

LOS ANGELES, CALIFORNIA 90025

(408) 720-8598

Attorney's Docket No. 005166.P006

"Express Mail" mailing label number: EL617183159US

Date of Deposit: August 10, 2001

I hereby certify that I am causing this paper or fee to be deposited with the United States Postal Service "Express Mail Post Office to Addressee" service on the date indicated above and that this paper or fee has been addressed to the Assistant Commissioner for Patents, Washington, D. C. 20231

Geneva Walls

(Typed or printed name of person mailing paper or fee)

(Signature of person mailing paper or fee)

August 10, 2001

(Date signed)

## COMMAND LINE INTERFACE ABSTRACTION ENGINE

### FIELD OF THE INVENTION

[0001] The present invention relates generally to field of command line interface application. More specifically, the present invention is directed to abstracting command line interfaces from embedded systems.

### BACKGROUND

[0002] An embedded system is a specialized computer in which operating system and application functions are often combined into the same program. Generally, an embedded system implies a fixed set of functions programmed into a non-volatile memory (ROM, flash memory, etc.) in contrast to a general-purpose computing machine.

[0003] Embedded systems are growing more and more complex, requiring a range of external management applications such as SNMP, Command Line Interfaces, TNM, WBEM, etc. These applications have completely different interfaces, network protocols and purposes but must all interface to the low-level internals of the embedded system.

[0004] In order for an embedded system to perform successfully in an environment, it is necessary that the embedded system be configured to operate as desired for troubleshooting and monitoring purposes. Further, the embedded system needs to be able to provide its current configuration or restore to a saved configuration when requested. This functionality is collectively known as "management" of the embedded system.

[0005] Management of the embedded system may be effected through a variety of user interfaces. The most common is Command Line Interface (CLI), where configuration and display commands are entered in text mode and promptly executed. Another common interface is Simple Network Management Protocol (SNMP) where configuration and monitoring data is transmitted over the network.

[0006] Prior technology for implementation of command line user interfaces relies on a set of closely coupled functions between the commands defined and the application functions or services of the embedded system. Typically a set of function calls is provided by the embedded system and utilized in the implementation of the CLI. This leads to a dependency of the user interface application on the underlying implementation of the embedded system which requires the embedded system services to provide a consistent interface to the CLI.

[0007] Since the user interface application and the application functions (or services) in the embedded system are inextricably linked through this functional interface (i.e., function calls), this leads to a brittle code structure that is not easily extended, difficult to manage and maintain. It would be advantageous to have a method that provides a generalized user interface to an embedded system in such a way that the generalized user interface is decoupled from the services and features delivered and implemented in the embedded system.

## SUMMARY OF THE INVENTION

[0008] In one embodiment, a method for abstracting a command line interface (CLI) from an embedded system is disclosed. A CLI description language is implemented to define a set of command keywords and arguments forming CLI description files. The CLI description language is implemented using Extensible Markup Language (XML). A set of XML tags and corresponding semantic definitions are provided. An XML schema is defined for the CLI description language. The CLI description files are compiled into run time format using the XML schema. The CLI description files provide a mapping of object names to services of the system.

FOOTNOTES

## BRIEF DESCRIPTION OF THE DRAWINGS

[0009] The present invention is illustrated by way of example in the following drawings in which like references indicate similar elements. The following drawings disclose various embodiments of the present invention for purposes of illustration only and are not intended to limit the scope of the invention.

[0010] **Figure 1** illustrates exemplary XML (Extensible Markup Language) tags used in XML CLI description files.

[0011] **Figure 2** is an exemplary illustration of CLI commands and a corresponding XML fragment.

[0012] **Figure 3** is an exemplary illustration of CLI commands with arguments and argument help and a corresponding XML fragment.

[0013] **Figure 4** illustrates an exemplary DTD schema used in accordance with the present invention.

[0014] **Figure 5** is an exemplary illustration of a Document Object Model (DOM) tree of an XML fragment describing a "clock set" command.

[0015] **Figure 6** is an exemplary illustration of a mapping from a XML CLI description file to a corresponding Java class CLIState.

[0016] **Figure 7** is an exemplary illustration of relationship among an XML description fragment, a CLI State and a CLI Action.

[0017] **Figure 8** is an exemplary illustration of the CLIBuilder generating ".cli" files to be interpreted by a CLIInterpreter.

[0018] **Figure 9** is an exemplary illustration of a CLI Interpreter.

[0019] **Figure 10** illustrates an exemplary CLIState tree.

[0020] **Figure 11** illustrates exemplary CLIState transitions for the CLIState tree in **Figure 10**.

[0021] **Figure 12** illustrates an exemplary block diagram of one embodiment of the design process to implement the CLI application.

[0022] **Figure 13** illustrates exemplary keyword sets for each of these commands.

[0023] **Figure 14** illustrates exemplary arguments for the "clock set" command.

[0024] **Figure 15** illustrates exemplary help strings for different command sequence.

[0025] **Figure 16** is an exemplary table illustrating CLI arguments to Java type mapping.

[0026] **Figure 17** illustrates an exemplary XML CLI description file for the "show clock" and "show clock detail" commands.

[0027] **Figure 18** illustrates an exemplary XML CLI description file for the “clock set” command.

[0028] **Figure 19** illustrates exemplary Java methods for the “clock set” and “clock set detail” commands.

[0029] **Figure 20** illustrates an exemplary Java method for the “clock set” command.

FIG. 20

## DETAILED DESCRIPTION

[0030] In one embodiment of the present invention, a method for abstracting a command line interface (CLI) application from an embedded system is disclosed. The abstraction separates high-level applications from the details of the embedded system. A CLI description language is implemented to define a set of command keywords and arguments forming CLI description files. The CLI description files are compiled to create run time modules. The runtime modules are provided as an application from the embedded system to interface applications.

[0031] The following detailed description sets forth numerous specific details to provide a thorough understanding of the invention. However, those of ordinary skill in the art will appreciate that the invention may be practiced without these specific details. In other instances, well-known methods, procedures, protocols, components, algorithms, and circuits have not been described in detail so as not to obscure the invention.

[0032] In this description, the CLI application refers to an application that allows for interaction with an embedded system (also referred to as a system or a platform). The CLI application is specified by a collection of modes, commands, keywords, and arguments that are used to command, control, and configure the systems.

[0033] Generally, a system is managed by using high-level applications that collect data from or send data to the system. The high-level applications may issue commands to the CLI application. In one embodiment of the present



invention, a framework is provided for abstracting the low-level elements of the system forming a CLI application. The frame work provides a system wide abstraction layer, below which are kernel device drivers and various system services software in the system and above which are the high-level applications that manage the system. Using this framework, the high-level applications can be insulated from the implementation of services on the system.

[0034] In one embodiment, architecture for the CLI application includes XML CLI description files, an XML compiler, a CLI interpreter, and a set of system interfaces. The XML CLI description files use XML syntax to describe features and structure of the CLI. This includes CLI modes, command, keyword and sub-keyword structures, help text for keywords and arguments, and specification of actions performed for a command. The XML compiler compiles the XML CLI description files into a format usable by a runtime system. The CLI interpreter is the runtime system responsible for presenting the CLI commands and for executing the actions specified by the CLI commands. The system interfaces includes a collection of Java interfaces (e.g., classes and methods). During runtime, the CLI commands are performed based on their semantics. The XML CLI description files, the XML compiler, the CLI interpreter, and the set of system interfaces form the framework to create the CLI application.

[0035] **Figure 1** illustrates exemplary XML (Extensible Markup Language) tags used in XML CLI description files. Each XML tag occurs in an XML CLI description file as a <tagname>...</tagname> combination. Attributes are enclosed in the tag fields with their values bounded by a pair of double quotation marks. In one embodiment, each of the XML CLI description files have the following form:

```

<?xml version="1.0" encoding="US-ASCII" ?>
<!DOCTYPE cli SYSTEM "cli.dtd">

<!--CLI Definitions here... -->

```

The first line is an XML declaration that specifies the type of the document as an XML document. The second line is a document type declaration that attaches the .XML file to a particular schema. In this case the schema is "cli.dtd". The second line also specifies that the top-level element is "cli". The remainder of the document includes the specific tags used to define the CLI constructs.

[0036] The <cli> tag 105 is the top-level element of the XML document. All other tags are sub tags of this top-level <cli> tag 105. This <cli> tag 105 has no attributes and text inside the <cli> tag 105 is ignored. The <mode> tag 110 contains the mode of the management command. There should be only one <mode> tag 110 in the XML document. The <mode> tag 110 has a single attribute which identifies the name of the mode such as, for examples, <mode name='global'> or <mode name="exec"> for global or executive mode.

[0037] The <command> tag 115 is the top-level tag for a single command. It has no attributes and any text inside the <command> tag 115 is ignored. The <command> tag 115 contains a single <keyword> tag such as, for example,

```

<command>
    <keyword text="show">
</keyword>
</command>.

```

The <keyword> tag 120 specifies the text of a command keyword in the command set of the CLI such as, for example "show", "enable" or "logout". The text is supplied by the single attribute "text". The keywords may include other <keyword> tags to specify sub keywords.

[0038] The <help> tag 125 specifies help information for the command / keyword. The <help> tag 125 has no attributes. The text enclosed by the <help> tag 125 defines the help text for the outlying tag, i.e. for a keyword or argument. One example of the help information follows:

```
<keyword text="show">  
<help>Show running system information </help>  
<keyword text="clock">  
</keyword>
```

[0039] The <action> tag 130 specifies that there is an executable action for the enclosing keyword. For example, the action specified within the <action> tag 130 is carried out for the associated command when the carriage return is pressed. The "object" and "method" attributes of the <action> tag 130 specify an object and a method in the system by name. A collection of Java type interfaces is created for the "objects" and "methods" that get invoked during run-time. Following is one example of the object and the method follows:

```
<action object="CLISystem" method="clock">  
</action>.
```

[0040] The <arg> tag 135 defines an argument type for an <action> tag 130. The <arg> tag 135 has no arguments and text in the <arg> tag 135 is ignored. A single <type> tag or <help> tag may be inserted in the body of the <arg> tag 135. Following is one example of the <arg> tag 135:

```
<arg>
    <type type="Java.lang.String"></type>
    <help>destination address or hostname</help>
</arg>
```

[0041] The <type> tag 140 describes a Java type for input to the invocation of the object, method pair specified by the <action> tag 130. The Java type name is entered as a "type" attribute of the <type> tag 140. Other text in the body of the <type> tag 140 other than those associated with the "type" attribute is ignored.

[0042] In this example, XML is used to describe the command language. XML is advantageous because there are a lot of available XML tools (e.g., editor tool, browser tool, etc.) for development. Although XML is used, one skilled in the art would recognize that other generalized markup languages might also be used to describe or map the management commands to the objects of the system, where the objects are low-level functions implemented in the system.

[0043] The CLI commands are generally in the following format:

Prompt> keyword<sub>1</sub> keyword<sub>2</sub> ... keyword<sub>n</sub> arg<sub>1</sub> ... arg<sub>n</sub>

For example, the commands may be:

Prompt> show clock detail, or

Prompt> show interfaces fast Ethernet 0/0.

The structure of the XML CLI description file expresses the format of the commands. The keyword tags embedding expresses the sequence above. Following is an example of a XML CLI description fragment for a command:

```
<command>
  <keyword text="keyword1">
    <keyword text="keyword2">
      ...
      <keyword text=keywordn">
        <action object="example" method="whatever">
          <!--argument arg1 -->
          <arg><type type="Java.lang.String"></arg>
          ...
          <!--argument argm -->
          <arg><type type="Java.lang.String"></arg>
        </keyword>
      ...
    </keyword>
  </keyword>
</command>
```

[0044] **Figure 2** is an exemplary illustration of CLI commands and a corresponding XML fragment. The CLI description fragment 210 may be part of a XML CLI description file. The CLI commands in this example include a “show clock” command 200 and a “show clock detail” command 205. The respective output is shown below each command. These commands 200, 205 are represented by the XML CLI description fragment 210 using XML tags described in **Figure 1**. The XML CLI description fragment 210 describes the “show clock” command using keywords, sub-keywords, help text, arguments, and specification of actions to be carried out.

[0045] The XML CLI description fragment 210 is presented in its source format and is normally compiled using an XML compiler so that it can be used by a run-time system. When a user or a management application issues a CLI command, the run-time system performs object mapping, method mapping and the arguments are provided to carry out the action. For example, when the command "show clock" command 200 is issued, the run-time system executes the action associated with an object "clock" and a method "show", shown on line 212. When the command SHOW "clock detail" 205 is issued, the run-time system executes the action associated with an object "clock" and a method "showdetail," as shown on line 215. The run-time system invokes a collection of Java system interfaces corresponding to the object and method specified in the action.

Note that the structure of the command's keyword and sub keywords is reflected in the nesting of the <keyword> tags in the XML CLI description. file In addition, any number of commands, keywords, sub keywords and help for each of the associated keywords may be nested and created by this syntax.

[0046] **Figure 3** is an exemplary illustration of CLI commands with arguments and argument help and a corresponding XML fragment. In this example, the command is a "clock set" command. The command 321 illustrates the "help" feature by specifying the "?" as part of the of command argument. This "help" feature is also illustrated in the command 322 and 323 to get additional information about the format (e.g., time, month, year) of the "clock set" command. The command 324 illustrates a complete "clock set" command with all the possible arguments specified.

[0047] The XML fragment 325 describes the "clock set" commands 321-324 using the XML tags described in **Figure 1**. The XML fragment 325 describes the hierarchy of the CLOCK SET command keyword by keyword. At run-time, the command CLOCK SET issued by the user or the management application is recognized by the CLI interpreter as corresponding to the command keyword CLOCK shown on line 327 and the keyword SET shown on line 328. There may be multiple command keywords in the same XML document in addition to the CLOCK command. The CLI interpreter will parse the XML fragment 325 to determine if the command is a valid command. In the current example, the help information requested in the commands 321-323 is enclosed in the <help> tags on lines 330, 335, 340 and 345 respectively.

[0048] Depending on the format of each of the CLOCK SET commands 321-324, appropriate data is extracted from the arguments of the XML fragment 325 and is pushed to system. The action specified by the "object" and "method" attribute of the <action> tag shown on line 329 is performed by the system.

[0049] A Document Type Definition (DTD) is used to provide formal description of the structure of the XML CLI description file in terms of XML tags described in **Figure 1**. A DTD is a file (or several files to be used together), which contains a formal definition of a particular type of document (e.g., the XML CLI description document). It sets out a schema of names that can be used for element types, where they may occur, and how they all fit together. The format of the DTD is known to one skilled in the art. In one embodiment, the DTD is used in this invention to provide notice of what names and structures to be used for XML CLI description documents. Using the DTD, XML CLI description documents are ensured to be constructed and named in a conformant manner.

The DTD not only defines the valid XML CLI description files, but it can also be used by the tools to validate any input file before processing or by editors to assure that edited files conform to the DTD. **Figure 4** illustrates an exemplary DTD schema used in accordance with the present invention.

[0050] The XML tags and DTD schema described above represent the minimum required to implement the CLI application. It will be apparent to one skilled in the art that the set of XML tags, tag attributes, and the structure of the XML may be extended to accommodate other requirements or features required by the CLI application or other management tools. For example, XML tags may be added for documentation, web interface, or other management applications.

[0051] The set of CLI modes and commands may be described in XML CLI description files using the XML tags and the DTD schema. The XML CLI description files are translated by a Java tool called CLIBuilder into a format suitable to be interpreted at runtime by the CLI Interpreter (also referred to as a CLIEngine). In one embodiment, each XML CLI description file has a ".xml" suffix. The CLI Builder tools take the XML CLI description files and generate output files that are used as input to the CLI interpreter at runtime. In one embodiment, each output file has a ".cli" suffix. The CLIBuilder validates syntax and format of the XML CLI description files against the "cli.dtd" DTD schema. For example, the CLI Builder validates an XML CLI description file named "config.xml" and generates a "config.cli" file. The resulting "config.cli" file is to be inputted to the CLIEngine at runtime.

[0052] **Figure 5** is an exemplary illustration of a Document Object Model (DOM) tree of an XML fragment describing a "clock set" command. In one



embodiment, the CLIBuilder uses a Java DOM library to parse the input XML CLI description file into a DOM object model tree (or DOM tree). The DOM tree is a collection of Java objects that represent the tags, attributes, and text of an XML CLI description file. Each Java object is represented as a node in the DOM tree in **Figure 5**. The structure of the DOM tree shows relationship of the XML tags and text in the XML CLI description file. The structure also contains all of the text enclosed by any of the tags so that every piece of information in the XML CLI description file is reflected in the resulting DOM tree.

[0053] Each XML CLI description file is translated into the DOM tree format that can be interpreted at run-time. In one embodiment, a set of Java classes is used to provide for a more compact representation of the keywords and actions represented in the DOM tree. The CLIBuilder starts with the DOM tree and translates the DOM tree into a collection of Java runtime classes.

[0054] The DOM tree in **Figure 5** corresponds to the "clockset" command described in the CLI description fragment 225 illustrated in **Figure 2**. Note that there are two keyword nodes 505, 510 corresponding to the keywords "clock" and "set" respectively. Note also that there are four argument nodes 520-535 under the action node 515. The four argument nodes 520-535 correspond to the arguments for "time", "day of month", "month" and "year" of the "clock set" command. The node 540 contains text associated with the "help" node 538, which provides help information to the "arg" node 530.

[0055] **Figure 6** is an exemplary illustration of a mapping from a XML CLI description file to a corresponding Java class CLISState. The Java class CLISState is a construct that implements the CLI behavior based on the user input. It also

embodies the structure of the keywords and commands inherent in the XML CLI description files and in the DOM trees. To convert from the XML CLI description files and the DOM trees, the following exemplary attributes may be used for the CLIState class.

```
private String keyword
private String help
private AbstractMap substates
private CLIState superstate
private CLIAction action
```

Each CLIState has a keyword label, possible children substates, a parent state, and an action. This is analogous to the <keyword> tag in the XML CLI description file. The keyword is the label or text for the CLIState. This is the text attribute of the <keyword> tag. Referring to **Figure 6**, from the XML fragment 600, the "help" field 605 gets its content from the text of the <help> tag 610 in the XML fragment 600 of the enclosing <keyword> tag. Each of the keyword tags is represented by a CLIState. For example, the keyword "show" in the XML fragment 600 is represented by the CLIState 615, and the keyword "clock" is represented by the CLIState 620.

[0056] When a keyword tag encloses other keyword tags, then each of the CLIStates is a sub state of an enclosing CLIState. For example, the CLIState 620 is a sub state of the CLIState 615. Similarly, the CLIState 615 may be a sub state of another CLIState. A super state is a parent node in the CLIState tree for this CLIState. It is the enclosing keyword tag in the XML CLI description file. For example, the CLIState 615 is the super state for the CLIState 620. If there is no

enclosing keyword (i.e. the keyword is already a top-most keyword of a command), then a CLIState representing the mode of the command is the super state. The action is an action for the CLIState. If there is an action tag present in the XML CLI description file, this will be non-null.

[0057] To translate from the XML CLI description file to the CLIState structure, a CLIState is created for each <keyword> tag in the XML CLI description file. The mapping between the keyword and the CLIState is one-to-one. The keywords enclosed by another <keyword> tag are children (or sub states) of the enclosing CLIState. The <help> and <action> sub tags of a keyword are saved as values in the CLIState representing the keyword tag.

[0058] Figure 7 is an exemplary illustration of relationship among an XML description fragment, a CLI State and a CLI Action. In one embodiment, an <action> tag content in the XML CLI description fragment 700 is converted into a Java class CLIAction 720. Similar to the way the CLIState 710 embodies the information and structure of the modes, command, and keyword tags from the XML CLI description fragment 700, the Java class CLIAction 720 contains all the content of the <action> and the sub tags <type> and <help> from the XML CLI description fragment 700. To convert from the XML CLI description files and the DOM trees, the following exemplary attributes may be used for the CLIAction class.

```
private String object_name
private String method_name
private String [] argument_types
```

private String [] argument\_help

[0059] The "object\_name" 712 is the same as the value of the "Object" attribute of the <action>tag in the XML CLI description file. The "object\_name" is the name of the Java object that will be invoked when this action is performed. The "method\_name" 714 is the value of the "Method" attribute of the <action> tag in the XML CLI description fragment 700. This is the string of the method that will be invoked when this action is performed. The "argument\_types" 716 is the value of the "Type" attributes of the <type> tag in the XML CLI description fragment 700. The "argument\_types" is an array that is filled in order of the <arg> tags present in the action. The "argument\_help" 718 is the text of the <help> tags for each <arg> in the XML structure. The CLIState 710 and the CLIAction 720 embody the XML CLI description fragment 700 and the corresponding DOM tree representation. The structure of CLIStates and the CLIActions is then serialized by a CLIBuilder into a file to be read by the CLI Engine (or CLI Interpreter) for interpretation, as illustrated in **Figure 8**.

[0060] The runtime behavior of the CLI system is performed by multiple Java classes. The Java class CLIState implements the structure of the CLI commands keywords via transitions, as described above. The Java class CLIAction translates arguments from the input string and then invokes the object and method required for that input. The CLIEngine class initializes and keeps track of global states of the CLI and houses the object registry. The system interface class is the collection of Java objects that are invoked by the CLIActions.

[0061] **Figure 9** is an exemplary illustration of a CLI Interpreter. The Java class CLI Interpreter 900 is the main control structure of the CLI runtime and

performs housekeeping duties including initialization, object registration, action resolution and context data. During initialization, the CLI Interpreter 900 installs the CLIState trees 905 into the system. The CLIState trees 905 are internally installed and are associated with the CLI modes 910. The system interface 925 includes objects that are registered and associated with an object name in an object registry 915. This is the same name that is referenced by the actions in the CLIState tree or with the object attribute in the XML CLI description file. For action resolution, the CLIAction instances (in the CLIState trees) contain string references to the objects and methods that comprise the system interface 925. These are resolved into action objects and methods using the object registry 915 of the CLI Interpreter 900. Context data refers to all data that is global to the running of the engine itself and includes "current mode", "prompt" and "I/O stream". The CLI is generally in a "current" mode which corresponds to a particular CLIState tree that is active. A CLI prompt is formatted according to the current mode, and the CLI can take input from or output to (i.e., I/O stream) any kind of stream (e.g., a file or interactive input).

[0062] In one embodiment, the objects are registered in the system through a static method of the CLIEngine. For example, the method is

RegisterObject (Object, String).

This registration function should be done sometime during the initialization of the system. The method takes the object being registered and its "string name" (i.e., the name found in the XML CLI description file) as the object attribute of the action tag. For example, using the above method, the following registration can be done:

CLIEngine.registerObject (new InterfaceManager, "ethManager");

The object registry is used to resolve the static CLIAction information from the XML CLI description file to references to the actual objects and methods of the system interface.

[0063] As previously described, the CLIStates model the keyword tags of the XML CLI description files, their keyword labels, and the structure of possible substates and parent state (or super state). Recall that the CLIStates have the following attributes:

```
private String keyword;  
private String help  
private AbstractMap sub states;  
private CLIState super state;  
private CLIAction action.
```

The CLIStates also model behavior of the interfaces through the transition functions. In addition to the above method, another method is:

```
public void transition (StringTokenizer input);
```

where the "keyword" is the label or the text for the CLIState. During runtime, the valid input to transition to this state is from its parent. The "help" is displayed when the transition input is the help token "?". The "sub states" is an abstract map "M" such that:

$M : \text{String} \rightarrow \text{CLIState}.$

The keyword field of the sub states is the key set for this mapping. The "sub states" themselves are the resulting values. This allows for implementing the transition function. The "super state" is the upper node in the CLIState tree for this CLIState. The transition is the function that implements a walk of the CLIState tree based on the user input. For example, for an input token string  $k$ , CLIState  $p$  with sub state  $s$ , the transition function is:

$$p.\text{transition}(k) = s \text{ if and only if } (s.\text{keyword} = k)$$

The "action" indicates action for the CLIState. The action is invoked when it is at the end of the input and/or when the CLIState is a terminal node (i.e. no children). **Figure 10** illustrates an exemplary CLIState tree. The CLIState tree 1000 represents the "show clock" and the "show clock detail" commands. The different CLIStates 1005, 1010, 1015 are labeled in the CLIState tree 1000 in **Figure 10**. There may be other sub states under the initial state, show states, etc.

[0064] **Figure 11** illustrates exemplary CLIState transitions for the CLIState tree in **Figure 10**. Each CLI State represents a possible transition from an input token that matches its keyword. The transitions of the CLI State tree from the initial node down to the leaf are illustrated in **Figure 10**. The CLIState tree 1105 is similar to the CLIState tree 1000. The CLIState tree 1110 illustrates transition after encountering the token "show". The CLIState tree 1115 illustrates transition after further encountering the token "clock". The CLIState tree 1120

illustrates transition after further encountering the token "detail".

[0065] The CLIState tree is navigated based on the input tokens using the transition method of the current node in the CLIState tree. The transition function implements all of the runtime behavior of the CLIState tree, including navigation, command help, partial command help, unknown and incomplete command checks and whether an action should be invoked.

[0066] Upon receiving the serialized input (i.e., the serialized CLIState tree file described in **Figure 8**), the CLIAction object in the CLIState contains the static description of the object, method, and types of the arguments. The CLIAction invokes the specific action specified by the CLIState. This is basically the XML content described above, including:

```
private String oobject_name,  
private String method_name,  
private String [] argument_types, and  
private String [] argument_help.
```

Via the resolve method of the CLIAction, these strings are converted, using reflection and the input object\_map from the CLIEngine, into the actual object, method, and constructors that are used to invoke the method.

Consider the following exemplary action:

```
private Object object;  
  
private Method method;
```



```

private Constructor [] constructor;

public void resolve (AbstractMap object_map)
{...}

public void execute (StringTokenizer input)
{...}

```

where the "object" is the reference to the actual object that will be invoked when this action is executed. The "object\_name" field is used as the key to index into the input "object\_map". The "object\_map" is the object repository that is compiled by the CLI Interpreter. The "method" corresponds to the Method\_name of the CLIAction. The Method\_name may be retrieved using a "getMethod ()" method in the class of the objects described above. For each of the types in the "argument\_types" array, a "constructor" from "String" is retrieved and stored in the array in argument order. In this way arguments can be constructed from the tokens on the command line.

[0067] When a CLIState is reached such that an action should be performed (i.e. it is either a terminal node with a resolved action, or it is an input that has ended at a non terminal node with an action), an "execute ()" method of the CLIAction is called for that CLIState. This method marshals the remaining tokens into method arguments by calling the constructors in the "constructors []" array. The method is then invoked for the resolved object. The following example illustrates a resolved CLIAction instance. The left side is the CLIAction and the right side is the resolved "ethManager" instance.

```

object_name = "ethManager"
method_name = "show"
argument_types = {"java.lang.String"}
argument_help = {"<0-6> Fast..."}
    object
    method
    constructor = { String(String), }

```

```

public void show (String id) { }

```

**[0068]** Figure 12 illustrates an exemplary block diagram of one embodiment of the design process to implement the CLI application. The design process integrates and implements CLI commands in the framework described herein. The process starts at block 1200. At block 1205, the CLI commands are defined. For example, this may include defining command input and output requirements, assignment of the mode for the command, defining the set of keywords and sub keywords for the command, defining the argument set for each action of the command (this may include a consideration of the Java classes that will be constructed from the string input), defining help texts for each keywords and argument, etc.

**[0069]** At block 1210, the system interface is designed and implemented. This includes defining the Java classes and methods with each action specified in the command definition having an object and method associated with them to perform the semantics required by the command. This also includes defining the arguments where each argument of the specified method having a constructor from "String" that can be invoked, and defining object registration to establish names for each object instance for mapping in XML.

**[0070]** At block 1215, the XML CLI description files are created. The XML CLI description files may be written and added directly to the CLIBuilder after the

command definition and system interface design are complete. At block 1220, the new ".xml" XML CLI description files are built using the CLIBuilder to create ".cli" files. The resulting ".cli" files may then be downloaded to the system and tested. The design process ends at block 1225.

[0071] Following is an example of designing a CLI application using the design process illustrated in the flow diagram in **Figure 12**. The example is described using the commands "show date", "show date detail", and "set clock".

Following are exemplary desired input and output formats for these commands:

```
input> show clock
output> 04:10:12.621 UTC Thu May 6 1993
```

```
input> show clock detail
output> 04:10:28.109 UTC Thu May 6 1993
        Time source is user configuration
```

```
input> clock set ?
output> hh:mm:ss Current Time
```

```
input> clock set 00:00:00 ?
output> <1-31> Day of the month
```

```
input> clock set 00:00:00 1 ?
output> MONTH Month of the year
```

```
input> clock set 00:00:00 25 Sep ?
output> <1993-2035> Year
```

```
input> clock set 00:00:00 31 Feb 2000
output> Invalid date (doesn't exist)
```

```
input> clock set 00:00:00 25 Sep 2000
output>
```

[0072] The above exemplary commands reside in an "executive" mode of the CLI. The "show date" and "show date detail" commands are also in the global mode or user mode so the same XML text to describe the commands are put in

XML CLI description file for the global mode and for the user mode. **Figure 13** illustrates exemplary keyword sets for each of these commands. For example, for the command "show date detail" 1305, the corresponding keyword set 1310 include the keywords "show", "date" and "detail". In the current example, the "show date" and "show date detail" commands do not have arguments. However, the "clock set" command has several arguments. **Figure 14** illustrates exemplary arguments for the "clock set" command.

[0073] **Figure 15** illustrates exemplary help strings for different command sequence. It is suggested that each command keyword have a help message. In the current example, following are some help messages for the clock command.

```
Prompt > ?
...
clock      Manage the system clock
...
show       Show running system information
...
Prompt > show ?
...
clock      Display the system clock
...
Prompt > show clock detail?

clock      Display detailed information

Prompt > clock ?

set        Set the time and date
```

[0074] The following descriptions involve system interface design and implementation. This includes Java class definition, method definition and argument mapping. To implement the command definition, a Java class is

written having methods that can be invoked by the CLI to perform the required semantics. For example, for the exemplary commands described above, a single Java class called "clock" is created with the following methods:

```
public class clock {  
  
    /** Display the system clock */  
    public void show ()  
  
    /** Display detailed information */  
    public void showDetail ()  
  
    /** Set the time and date */  
    public void set (String time, Integer day, String month,  
                    Integer year)
```

[0075] Figure 16 is an exemplary table illustrating CLI arguments to Java type mapping. The table list argument number 1605, the argument description 1610 and the Java type 1615 for the arguments of the "clock set" command. Using the "clock set" command example, to implement the command via the description of the arguments, each argument is converted from the input string into a selected Java type 1615. Each of the selected Java types 1615 is an existing Java class that has an existing constructor "Java.lang.String" 1620. If a user-defined class is used, a constructor from "Java.lang.String" 1620 needs to be provided in order for the CLI to be able to construct and marshal the arguments from the input to the method.

[0076] Given the class definition described above, at some point during the initialization of the system an instance of the class will be created. In the current example, the top-level CLI object will create a "Clock" instance during startup,

then register (i.e., object registration) the instance with the CLI with the name "Clock", such as, for example:

```
CLIEngine.registerObject ("Clock", new Clock ()).
```

[0077] Once the commands are defined, the system interface methods are defined and implemented, and the instance is named and registered, the full XML CLI description files for the commands can be written. **Figure 17** illustrates an exemplary XML CLI description file for the "show clock" and "show clock detail" commands. **Figure 18** illustrates an exemplary XML CLI description file for the "clock set" command.

[0078] **Figure 19** illustrates exemplary Java methods for the "clock set" and "clock set detail" commands. The methods output the current date to the output stream in the proper format. Using the Java data formatting classes greatly simplifies the effort. The "clock set" command is a little more complex because it includes arguments. The presence of the arguments to the method requires additional processing for the method. There are two parts to this processing, validating the arguments and performing the required action.

[0079] Argument validation comes in two forms, individual validation and multiple argument consistency. A command may require either, both or none of these validations. For example consider the day of the month argument for the "clock set" command. A string input to this argument that cannot be parsed into an integer is invalid. The expected input also must be in the range from 1 to 31 to be considered a valid input to this argument. This is an example of individual argument validation. The day of the month value of "31" will be

individually validated by the above definition but makes sense only for certain values of the next argument (i.e., the month). For example, "February 31" is obviously an invalid date. This is an example of multiple argument consistency. In one embodiment, individual argument consistency checks can be done at two levels, by the constructor for the input argument or by the method itself, while multiple argument consistency checks are performed by the method.

[0080] In one embodiment, argument correctness violations are handled in the method by throwing an `ArgumentException` instance. The number of the offending argument and position of the error in the argument can be set in the `ArgumentException`. The CLI interpreter responds to the exception with an output making the invalid argument. The following example illustrates the invalid argument and the output marking '^' by the CLI application.

```
Prompt> clock set 00:00:00 ggg Feb 2000
```

```
^
```

```
%Invalid input detected at the '^' marker
```

[0081] **Figure 20** illustrates an exemplary Java method for the "clock set" command. Again the use of the Java language greatly simplifies the implementation.

[0082] Using the method of the present invention, commands that are associated with the system to control the system can be shielded from the user. These commands can be described and expressed using a generalized markup up language (e.g., XML) as a description language. A set of commands is defined by describing all of the commands to be used with the system. The commands

are mapped to the services available in the system using objects and methods defined in Java classes. Using this method also shield the users from changes or version upgrades applied to the operating software of the system. In addition, when a new system is used, the commands can be made to be the same by describing and mapping the commands to the objects and methods of the new system.

[0083] The operations of the various methods of the present invention may be implemented by a processing unit in a digital processing system, which executes sequences of computer program instructions which are stored in a memory which may be considered to be a machine readable storage media. The memory may be random access memory, read only memory, a persistent storage memory, such as mass storage device or any combination of these devices. Execution of the sequences of instruction causes the processing unit to perform operations according to the present invention. The instructions may be loaded into memory of the computer from a storage device or from one or more other digital processing systems (e.g. a server computer system) over a network connection. The instructions may be stored concurrently in several storage devices (e.g. DRAM and a hard disk, such as virtual memory). Consequently, the execution of these instructions may be performed directly by the processing unit.

[0084] In other cases, the instructions may not be performed directly or they may not be directly executable by the processing unit. Under these circumstances, the executions may be executed by causing the processor to execute an interpreter that interprets the instructions, or by causing the processor to execute instructions which convert the received instructions to



instructions which can be directly executed by the processor. In other embodiments, hard-wired circuitry may be used in place of or in combination with software instructions to implement the present invention. Thus, the present invention is not limited to any specific combination of hardware circuitry and software, nor to any particular source for the instructions executed by the computer or digital processing system.

[0085] Although the present invention has been described with reference to specific exemplary embodiments, it will be evident that various modifications and changes may be made to these embodiments without departing from the broader spirit and scope of the invention as set forth in the claims. Accordingly, the specification and drawings are to be regarded in an illustrative rather than a restrictive sense.